The Clean Code Blog

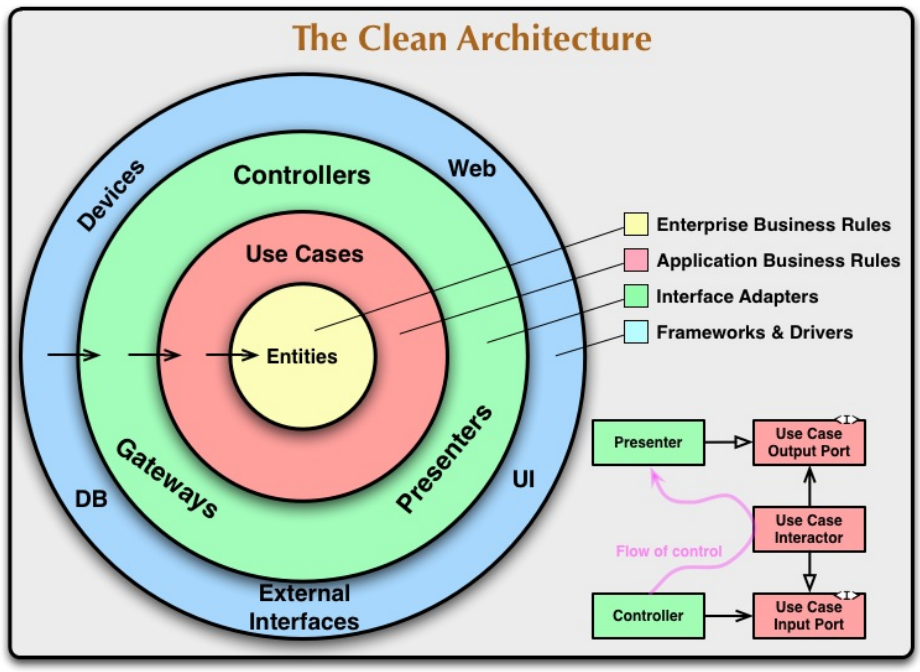
by Robert C. Martin (Uncle Bob)

atom/rss feed

- Functional Classes in Clojure
  01-19-2023
- Functional Classes
  01-18-2023
- Space War
  11-28-2021
- Functional Duplications
  10-28-2021
- Roots
  09-25-2021
- More On Types
  06-29-2021
- On Types
  06-25-2021
- if-else-switch
  03-06-2021
- Pairing

The Clean Architecture

13 August 2012



Over the last several years we've seen a whole range of ideas regarding the architecture of systems. These include:

- Hexagonal Architecture (a.k.a. Ports and Adapters) by Alistair Cockburn and adopted by Steve Freeman, and Nat Pryce in their wonderful book Growing Object Oriented Software
- Onion Architecture by Jeffrey Palermo

- Screaming Architecture from a blog of mine last year
- DCI from James Coplien, and Trygve Reenskaug.
- BCE by Ivar Jacobson from his book *Object Oriented Software Engineering: A Use-Case Driven Approach*

Though these architectures all vary somewhat in their details, they are very similar. They all have the same objective, which is the separation of concerns. They all achieve this separation by dividing the software into layers. Each has at least one layer for business rules, and another for interfaces.

Each of these architectures produce systems that are:

1. Independent of Frameworks. The architecture does not depend on the existence of some library of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
2. Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.
3. Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
4. Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
5. Independent of any external agency. In fact your business rules simply don’t know anything at all about the outside world.

The diagram at the top of this article is an attempt at integrating all these architectures into a single actionable idea.

The Dependency Rule

The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.

The overriding rule that makes this architecture work is *The Dependency Rule*. This rule says that *source code dependencies* can only point *inwards*. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the an inner circle. That includes, functions, classes. variables, or any other named software entity.

By the same token, data formats used in an outer circle should not be used by an inner circle, especially if those formats are generate by a framework in an outer circle. We don’t want anything in an outer circle to impact the inner circles.

Entities

Entities encapsulate *Enterprise wide* business rules. An entity can be an object with methods, or it can be a set of data structures and functions. It doesn’t matter so long as the entities could be used by many different applications in the enterprise.

If you don’t have an enterprise, and are just writing a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes. For example, you would not expect these objects to be affected by a change to page navigation, or security. No operational change to any

particular application should affect the entity layer.

Use Cases

The software in this layer contains *application specific* business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their *enterprise wide* business rules to achieve the goals of the use case.

We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI, or any of the common frameworks. This layer is isolated from such concerns.

We *do*, however, expect that changes to the operation of the application *will* affect the use-cases and therefore the software in this layer. If the details of a use-case change, then some code in this layer will certainly be affected.

Interface Adapters

The software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the Database or the Web. It is this layer, for example, that will wholly contain the MVC architecture of a GUI. The Presenters, Views, and Controllers all belong in here. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views.

Similarly, data is converted, in this layer, from the form most convenient for entities and use cases, into the form most convenient for whatever persistence framework is being used. i.e. The Database. No code inward of this circle

should know anything at all about the database. If the database is a SQL database, then all the SQL should be restricted to this layer, and in particular to the parts of this layer that have to do with the database.

Also in this layer is any other adapter necessary to convert data from some external form, such as an external service, to the internal form used by the use cases and entities.

Frameworks and Drivers.

The outermost layer is generally composed of frameworks and tools such as the Database, the Web Framework, etc. Generally you don't write much code in this layer other than glue code that communicates to the next circle inwards.

This layer is where all the details go. The Web is a detail. The database is a detail. We keep these things on the outside where they can do little harm.

Only Four Circles?

No, the circles are schematic. You may find that you need more than just these four. There's no rule that says you must always have just these four. However, *The Dependency Rule* always applies. Source code dependencies always point inwards. As you move inwards the level of abstraction increases. The outermost circle is low level concrete detail. As you move inwards the software grows more abstract, and encapsulates higher level policies. The inner most circle is the most general.

Crossing boundaries.

At the lower right of the diagram is an example of how we cross the circle boundaries. It shows the Controllers and Presenters communicating with the Use Cases in the next layer. Note the flow of control. It begins in the controller,

moves through the use case, and then winds up executing in the presenter. Note also the source code dependencies. Each one of them points inwards towards the use cases.

We usually resolve this apparent contradiction by using the Dependency Inversion Principle. In a language like Java, for example, we would arrange interfaces and inheritance relationships such that the source code dependencies oppose the flow of control at just the right points across the boundary.

For example, consider that the use case needs to call the presenter. However, this call must not be direct because that would violate *The Dependency Rule*: No name in an outer circle can be mentioned by an inner circle. So we have the use case call an interface (Shown here as Use Case Output Port) in the inner circle, and have the presenter in the outer circle implement it.

The same technique is used to cross all the boundaries in the architectures. We take advantage of dynamic polymorphism to create source code dependencies that oppose the flow of control so that we can conform to *The Dependency Rule* no matter what direction the flow of control is going in.

What data crosses the boundaries.

Typically the data that crosses the boundaries is simple data structures. You can use basic structs or simple Data Transfer objects if you like. Or the data can simply be arguments in function calls. Or you can pack it into a hashmap, or construct it into an object. The important thing is that isolated, simple, data structures are passed across the boundaries. We don't want to cheat and pass *Entities* or Database rows. We don't want the data structures to have any kind of dependency that violates *The Dependency Rule*.

For example, many database frameworks return a

convenient data format in response to a query. We might call this a RowStructure. We don't want to pass that row structure inwards across a boundary. That would violate *The Dependency Rule* because it would force an inner circle to know something about an outer circle.

So when we pass data across a boundary, it is always in the form that is most convenient for the inner circle.

Conclusion

Conforming to these simple rules is not hard, and will save you a lot of headaches going forward. By separating the software into layers, and conforming to *The Dependency Rule*, you will create a system that is intrinsically testable, with all the benefits that implies. When any of the external parts of the system become obsolete, like the database, or the web framework, you can replace those obsolete elements with a minimum of fuss.

Pliers

11-01-2015

- Future Proof

  10-30-2015

- Agile is not now, nor was it ever, Waterfall.

  10-16-2015

- VW

  10-14-2015

- WATS Line 54

  10-05-2015

- A Little Structure

  09-23-2015

- Make the Magic go away.

  08-06-2015

- Pattern Pushers

  07-05-2015

- The Little Singleton

  07-01-2015

- The First Micro-service Architecture

  05-28-2015

- Language Layers

  04-27-2015

- Does

Organization Matter?

04-15-2015

- The MODE-B Imperative

  02-21-2015

- They Called them Computers.

  02-19-2015

- 'Interface' Considered Harmful

  01-08-2015

- The Cycles of TDD

  12-17-2014

- OO vs FP

  11-24-2014

- Thorns around the Gold

  11-19-2014

- The Obligation of the Programmer.

  11-15-2014

- One Hacker Way!

  11-12-2014

- Laughter in the male dominated room.

  10-26-2014

- GOML-1, Responsive Design

  10-08-2014
- Clean Micro-service Architecture

  10-01-2014
- Microservices and Jars

  09-19-2014
- The More Things Change...

  09-18-2014
- Test Time

  09-03-2014
- A Little About Patterns.

  06-30-2014
- My Lawn

  06-20-2014
- Is TDD Dead? Final Thoughts about Teams.

  06-17-2014
- First

  05-19-2014
- The Little Mocker

  05-14-2014
- The Open Closed

Principle

05-12-2014

- Framework Bound[2]

  05-11-2014

- When to Mock

  05-10-2014

- The Single Responsibility Principle

  05-08-2014

- Professionalism and TDD (Reprise)

  05-02-2014

- Test Induced Design Damage?

  05-01-2014

- When TDD doesn't work.

  04-30-2014

- Monogamous TDD

  04-25-2014

- Code Hoarders

  04-03-2014

- The *True* Corruption of Agile

  03-28-2014

- When Should You Think?

03-11-2014

- A Spectrum of Trust

02-27-2014

- Oh Foreman, Where art Thou?

02-23-2014

- Where is the Foreman?

02-21-2014

- The Domain Discontinuity

01-27-2014

- Coding in the Clink (9)

01-20-2014

- Extreme Programming, a Reflection

12-10-2013

- Novices. A Coda

11-25-2013

- Hordes Of Novices

11-19-2013

- Healthcare.gov

11-12-2013

- The Careless Ones

10-24-2013

- Dance you Imps!

10-01-2013

- A.T. FAIL!

  09-26-2013

- Test First

  09-23-2013

- Transformation Priority and Sorting

  05-27-2013

- The Transformation Priority Premise

  05-27-2013

- Flash - TPP

  05-27-2013

- Fib. The T-P Premise.

  05-27-2013

- There are Ladies Present

  03-22-2013

- The Frenzied Panic of Rushing

  03-11-2013

- An Open and Closed Case

  03-08-2013

- The Pragmatics of TDD

  03-06-2013

- The Start-Up Trap

  03-05-2013
- The Principles of Craftsmanship

  02-10-2013
- The Humble Craftsman

  02-01-2013
- The Laborer and the Craftsman

  01-30-2013
- FP Basics E4

  01-29-2013
- FP Basics E3

  01-07-2013
- FP Basics E2

  01-02-2013
- Brave New Year

  12-29-2012
- FP Basics E1

  12-22-2012
- Three Paradigms

  12-19-2012
- The New CTO

  09-06-2012
- Functional Programming for the Object Oriented

Programmer

08-24-2012

- The Clean Architecture

  08-13-2012

- NO DB

  05-15-2012

- Why is Estimating so Hard?

  04-20-2012

- After the Disaster

  04-18-2012

- Service Oriented Agony

  02-01-2012

- The Ruby Colored Box

  01-31-2012

- Fecophiles

  01-20-2012

- The Letter

  01-12-2012

- Flipping the Bit

  01-11-2012

- The Barbarians are at the Gates

  12-11-2011

- Clean Architecture

11-22-2011

- Double Entry Bookkeeping Dilemma. Should I Invest or Not?

  11-06-2011

- Simple Hickey

  10-20-2011

- Screaming Architecture

  09-30-2011

- Bringing Balance to the Force

  01-19-2011

- What Software Craftsmanship is about

  01-17-2011